COL215 Assignment-3 Report

Ashish Arora (2021CS10069) and Aaveg Jain (2021CS10073)

Problem Description
Given a function with 0s, 1s, and x's for each input combination, print the minimised sum of product terms, where these terms are obtained by deleting as many as terms as possible from the expanded function of the input combination, so that the number of terms thus obtained, is minimised

Our Approach-

Functions

1. join(arr)

   This function takes a list of terms "arr" as a parameter. It iterates through the elements of the list and add the element to the end of a string if its not None. This string is then returned. Example: ["a", "b", None, "d'"] returns "abd'"

2. parse(string)

   This function takes "string" as a parameter. It generates a list of literals from the given string and assigns None if the string does not contain a possible literal. Example: "abc" returns ["a","b","c",None] for a kmap with 4 literals

3. parse2(string)

   This function takes a string "string" as a parameter. It generates a list of literals from the given string. Example: "abcd'" returns ["a", "b", "c", "d'"]

4. findmaximal(one, dontcare)

   This is the main required function. It takes the lists one and dontcare as its parameters. The lists one and dontcare contain the terms corresponding to one and dontcare. The elements in these lists are then parsed using the parse [2] function and then added to a combined list. The list of parsed list of one is then iterated, for each of the elements a DigitalTree [5] object is initialised and then the max element (corresponding to the maximum region) is found out using the find_max function of the DigitalTree object. This region is then stored in the ans list (which stores the maximum region corresponding to each element in one)

5. islegal(terms)

   This function takes a list of terms as a parameter and returns whether the region (depicted by the list) is a valid region or not. It finds the no of terms enclosed by the region by finding out the no. of None terms present in the list and then raising 2 to its power. Now we initialise a count variable to 0. The function then iterates through all the elements of the combined list (check [3]) and checks whether the following term lies within our region, if it does then the value of count is increase by one.

After we are done iterating through all the elements of the combined list or count is equal to the number of terms in our region (whichever occurs earlier) we exit the loop. Then at the end we check whether count == no. of terms in our region. If it does, which means that all terms in our region are present in the combined list and hence valid, we return True otherwise False is returned.

6. Class Node:

The various functions of this class are:

a. __init__(self, term)

Initialises an object of class Node and assigns attributes term, no_of_var and a list children to it

b. __str__(self)

Returns the term joined in string form using the join function [1]

7. Class DigitalTree

The various function of this class are explained as follows:

a. __init__(self,term)

Intiailises an object of this class. Assigns attributes term, no_of_var, and tree

b. make_tree(self, arr)

This function iterates through the list arr and checks if an element is not None. If such an element is found then it temporarily replaces the value at this index to None and checks whether this new list forms a valid region. If it does then it calls make_tree function on this new list. Essentially creating a recursive function with each recursion call trying to expand the current region. After this is done it replaces the value at this index from None to the original value.

c. find_max_helper(self, tree)

This function recursively calls the children of the term passed as an argument and checks whether the term in argument is smaller than its child term. If that is the case then the term is updated and the term corresponding to the maximum region is returned

d. find_max(self)

This function calls the function find_max_helper on the tree associated with the object

8. Opt_function_reduce() –

Find_maximal() is used to find the maximal term of each cell in func_TRUE. Then duplicates are removed from the returned list and finally it is sorted in ascending order of size of regions(final list is reduced_answer). Then the generate_terms() function is used to generate all the terms of each region and added to a dictionary dict is it not already present. Otherwise the corresponding region is added to the value set of that term. After the dictionary is made, again for each region all the terms are generated using generate_terms() and checked if it is enclosed by more than one region. If this is true for all cells of the region, then the region is deleted from the dictionary by removing it from the value set of all the corresponding terms, otherwise if it not to be deleted then it is appended to the final answer (final_answer). Dontcares are ignored in this process.

HOW THE ALGORITHM WORKS

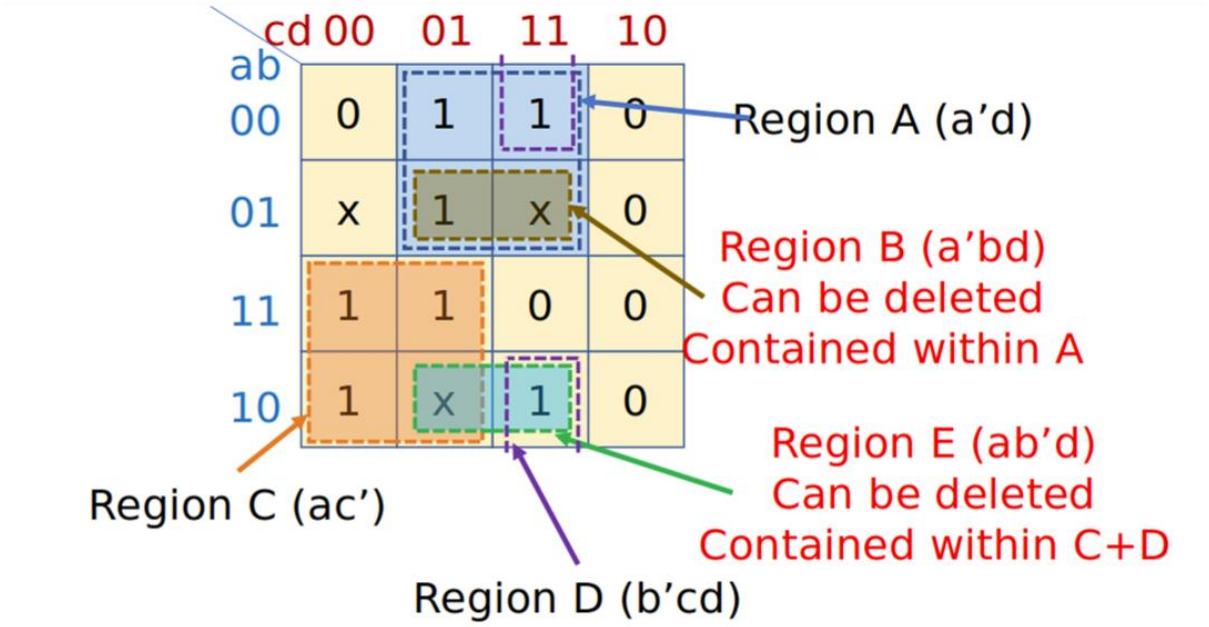
The algorithm has two parts – 1. Finding a maximally expanded region for each minterm in FUNC_TRUE.

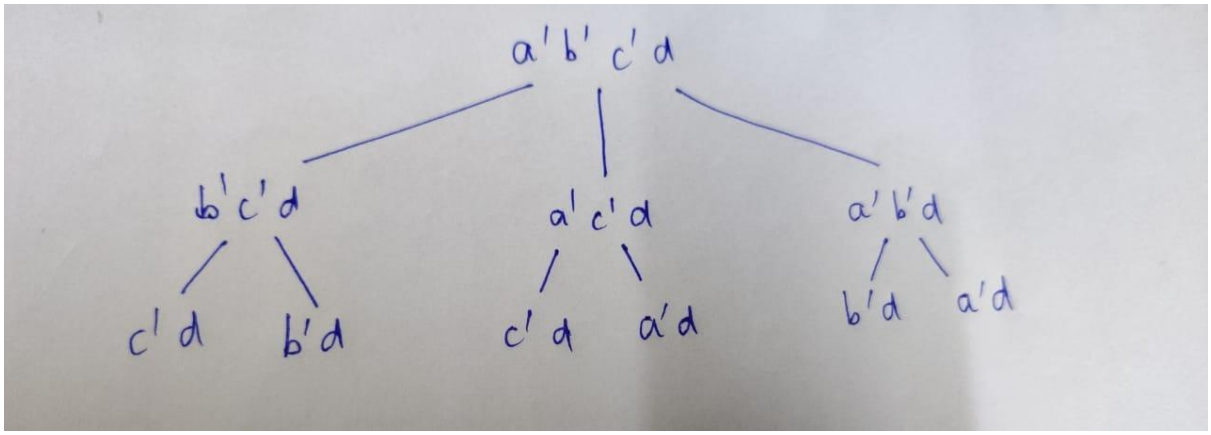
2. checking each cell of a maximally expanded region whether it is enclosed in another maximally expanded region. If it is, then the region is deleted.

1. HOW PART ONE WORKS

To maximally expand a cell, we make an abstract tree representing all valid expansions of the cell. For this we have made a class called DigitalTree which has nodes of class Node. Each Node has a list called 'children' which is a list containing references to each child of that node. The node also stored the term it represents.

What does it mean for a parent to have a child- consider the term "a'b'c'd" which has to be maximally expanded. On removing literal a', we get b'c'd. if b'c'd is also a valid region, then it made a child of a'b'c'd. this same process is done for each literal. This above process is achieved by a function called make_tree in the DigitalTree class, which is called in its constructor. For eg- consider the kmap below and the corresponding tree for term "a'b'c'd".





2. HOW PART TWO WORKS-

We first call the function find_maximal(func_TRUE,func_DC) to generate the maximal region for each term as outlined above. We then eliminate the duplicates and sort the answer so that smaller regions occur before larger regions and are thus deleted first.

We then make a dictionary with each cell in func_TRUE and func_DC as a key. The value of the key is a set of all the maximally expanded regions enclosing that term.

Then we start deleting the regions, smaller ones first. For each cell in the region to be checked for deletion, we first see if it is a dontcare. If it is, we ignore it since we have to cover all ones only. If the cell is a minterm (it is in func_TRUE), then we see the number of regions covering it. If it is one, then the region cannot be deleted and we go to the next region. If it is more than one, then we do the same process for the next cell in the region. If every minterm in the region is covered by atleast two regions, then that means every cell is covered by some other region (apart from the one we are currently considering) and thus the region can be deleted. During deletion we update our dictionary also so that the deleted region is removed from the value set of all the minterms it covered.

If the region cannot be deleted, then it is added to the final answer

TIME COMPLEXITY ANALYSIS-

Notation- n is the size of func_TRUE and func_DC combined. m is the number of literals in each cell.

We also assume n to be much larger than m for the purpose of asymptotic analysis.

The time complexities of all the functions are-

1. Join(arr) – O(m)
2. Islegal(terms) – O(mn)
3. Parse(string,n) – O(m)
4. Parse2(string) – O(m)
5. Make_tree(arr) – O(m*n!)
6. Find_max() – O(m*m!)
7. Find_maximal(one,dontcare) – O(n*m*m!)
8. Generate_terms(region,answer) – O(m * 2^m)
9. Opt_function_reduce(func_TRUE,func_DC) – O(n^2 * 2^m)
    a. Making the dictionary – O(n^2 * 2^m)
    b. Going through the dictionary and checking for each region to be deleted – O(n^2 *2^m)

Time complexity for each line can be found commented in the code for reference and for seeing how the above complexities were found.

Thus the overall complexity comes out to be O(n^2 * 2^m) (quadratic in the input size n and assuming n to be much larger than m)

TESTCASES RUN-

1. func_TRUE = ["a'b'c","a'bc"] , func_DC = ["ab'c"]
   this was run to see if all the cells of a region are being covered by some other region, but some dontcares aren't, in that case whether the region is deleted or not.
   This was successfully run as shown by the following output-

   OUTPUT-

   answer of comb_function_expansion is ["b'c", "a'c"]
   b'c to be deleted
   Term 1 : ab'c
   this term is a dont care, so ignored.
   Term 2 : a'b'c
   Covering region: a'c

   ["a'c"]

Cases where no deletion is occurring –

1. func_TRUE = ["a'b'c'd'e'", "a'bc'd'e'", "abc'd'e'", "ab'c'd'e'", "abc'de'", "abcde'", "a'bcde'", "a'bcd'e'", "abcd'e'", "a'bc'de", "abc'de", "abcde", "a'bcde", "a'bcd'e", "abcd'e", "a'b'cd'e", "ab'cd'e"]

   func_DC = []

   Output-

   ["c'd'e'", "cd'e", "abe'", 'bde', 'bc']

2. func_TRUE = ["a'b'c", "a'bc", "a'bc'", "ab'c'"]

   func_DC = ["abc'"]

   Output-

   ["bc'", "a'c", "ac'"]

Cases where deletion is occurring –

1. func_TRUE = ["a'b'c'd", "a'b'cd", "a'bc'd", "abc'd'", "abc'd", "ab'c'd'", "ab'cd"]

   func_DC = ["a'bc'd'", "a'bcd", "ab'c'd"]

   output-

   ["ac'", "b'd", "c'd"]

2. func_TRUE = ["a'bc'd'", "abc'd'", "a'b'c'd", "a'bc'd", "a'b'cd"]

   func_DC = ["abc'd"]

   output-

   ["a'b'd", "bc'"]

all cases ran successfully (checked with kmap). As cases where deletion is occurring or not occurring are run correctly, also the dontcares are being properly ignored and also the smaller terms are being deleted first(sorting is done), we can conclude the implementation is valid.